Taming DMA

Tales (of) Wrestling Memory Corruption

EMBEDDED
LINUX
CONFERENCE

Ahmad Fatoum – a.fatoum@pengutronix.de

About Me

Ahmad Fatoum

Pengutronix

a3f

a.fatoum@pengutronix.de

@a3f@fosstodon.org

- Kernel and Bootloader Porting
- Driver and Graphics Development
- System Integration
- Embedded Linux Consulting

Show of Hands

- Who has written or debugged driver code that uses DMA, whether in Linux or not?

Show of Hands

- Who has written or debugged driver code that uses DMA, whether in Linux or not?
- Who has done this on Linux in particular?

Show of Hands

- Who has written or debugged driver code that uses DMA, whether in Linux or not?
- Who has done this on Linux in particular?
- Who has done this in Linux userspace?

Show of Hands

- Who has written or debugged driver code that uses DMA, whether in Linux or not?
- Who has done this on Linux in particular?
- Who has done this in Linux userspace?
- Who has done this in Linux userspace, but actually used kernel mechanisms like VFIO for IOMMU control to make this safer?

Show of Hands

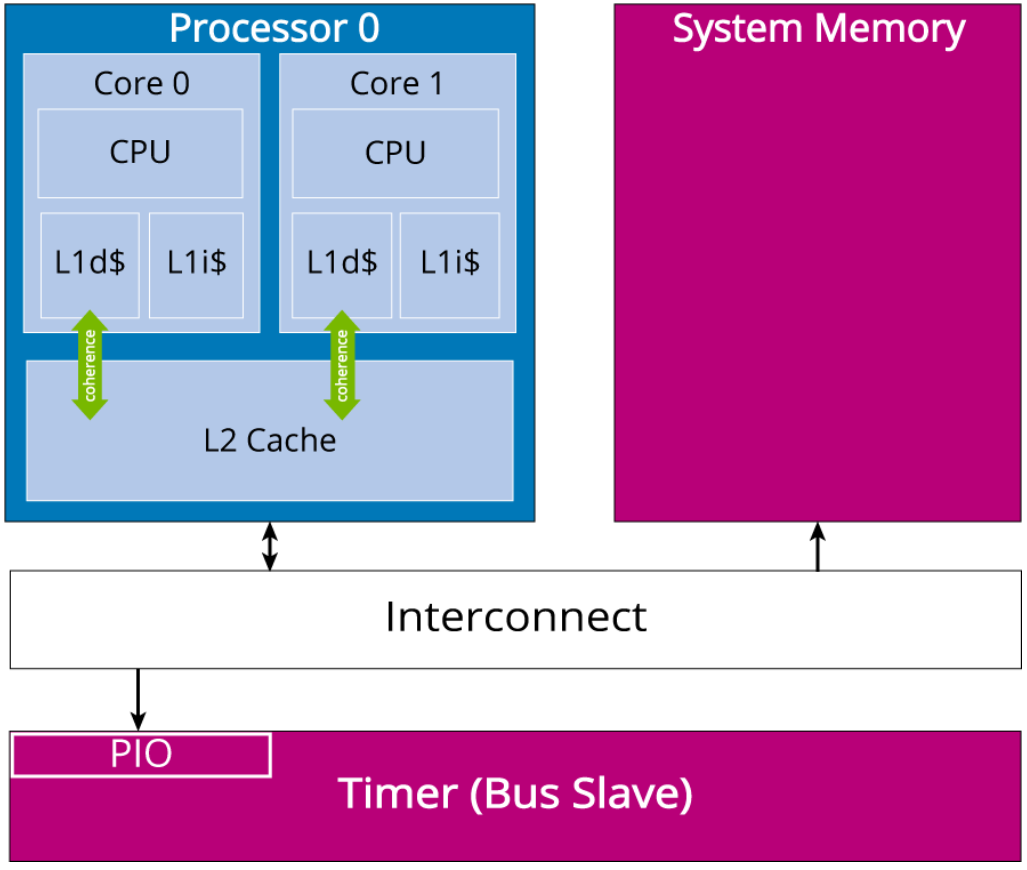
- Who has written or debugged driver code that uses DMA, whether in Linux or not?
- Who has done this on Linux in particular?
- ~~Who has done this in Linux userspace?~~
- ~~Who has done this in Linux userspace, but actually used kernel mechanisms like VFIO for IOMMU control to make this safer?~~

(just curious)

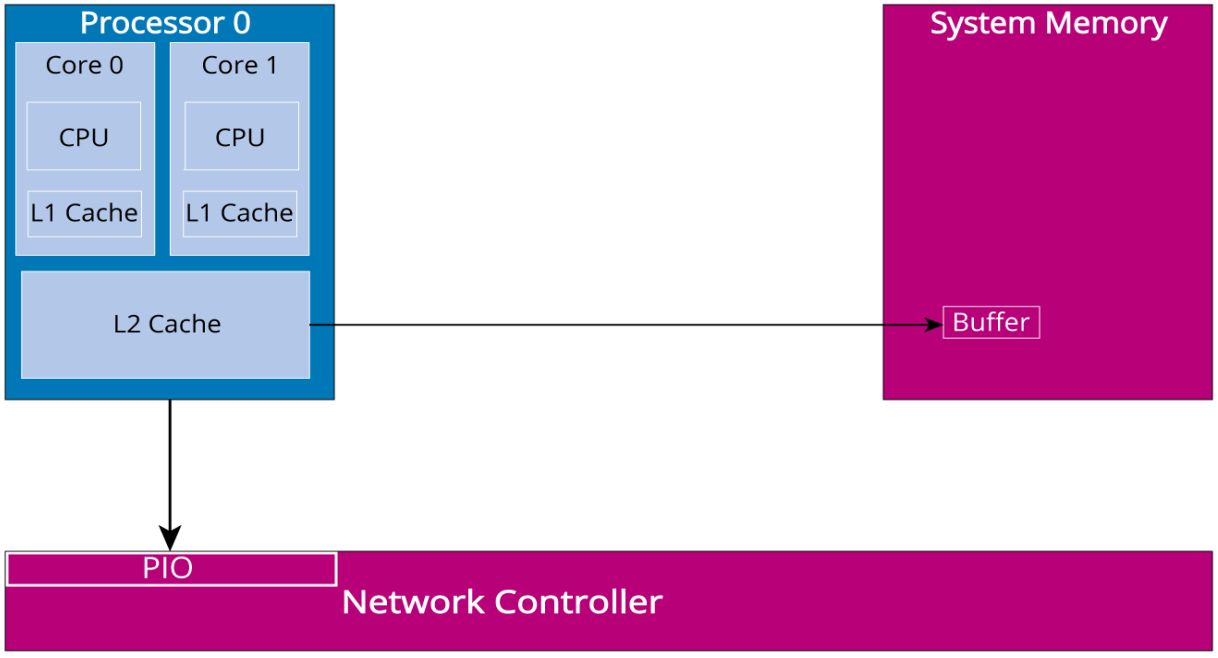
DMA: Direct Memory Access

A short Intro

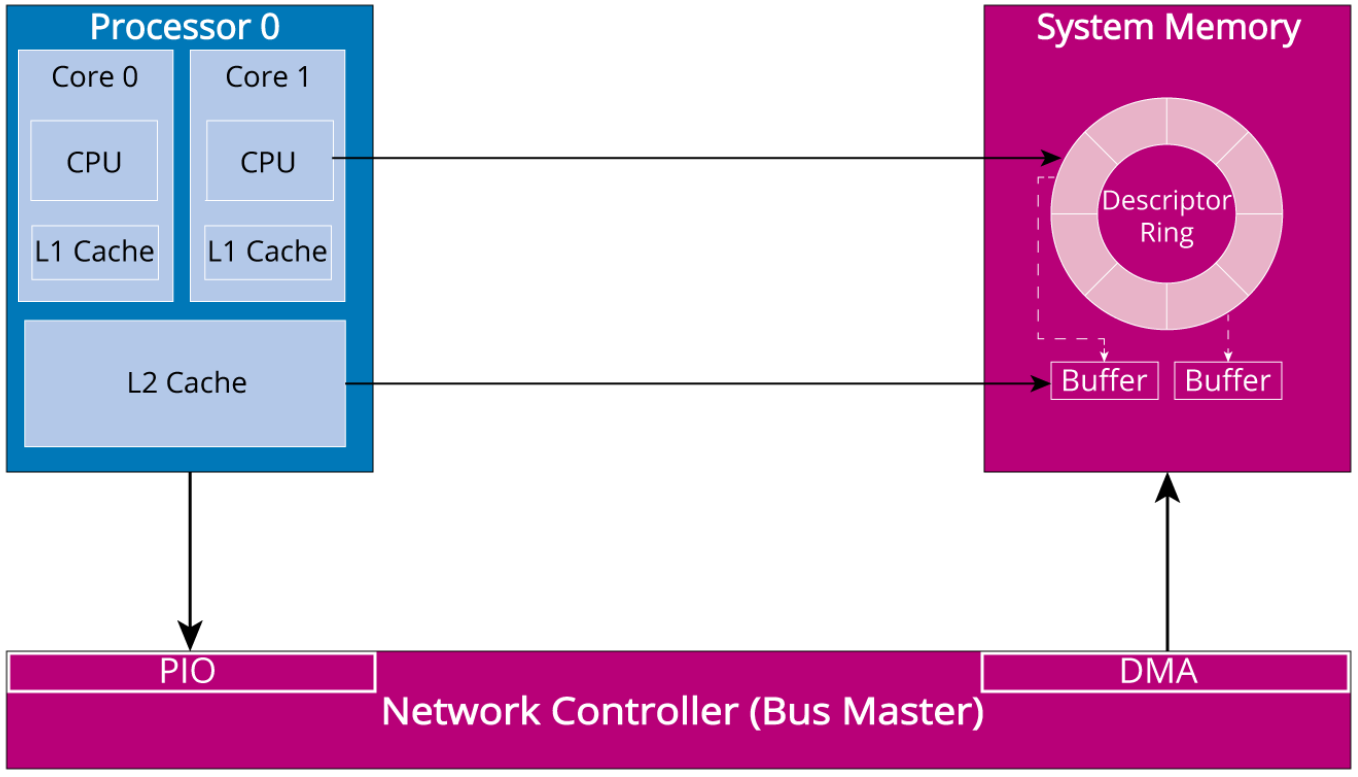
PIO: Programmed I/O



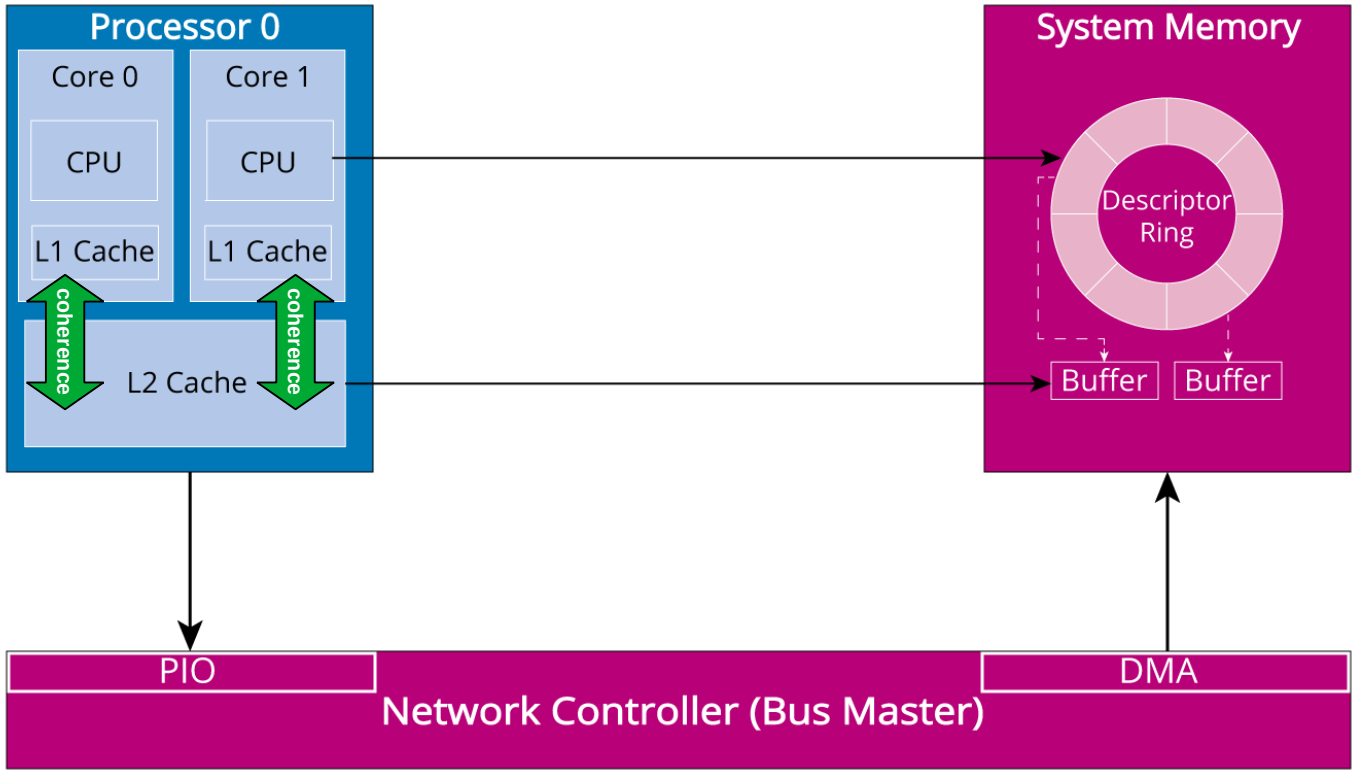
PIO: Programmed I/O



DMA: Direct Memory Access



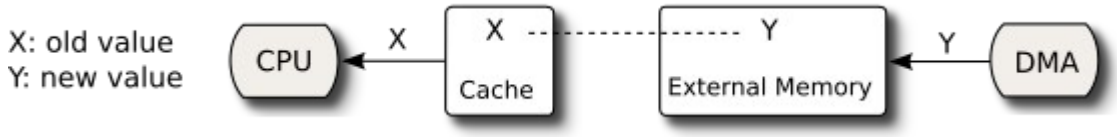
DMA: Cache Coherency Woes



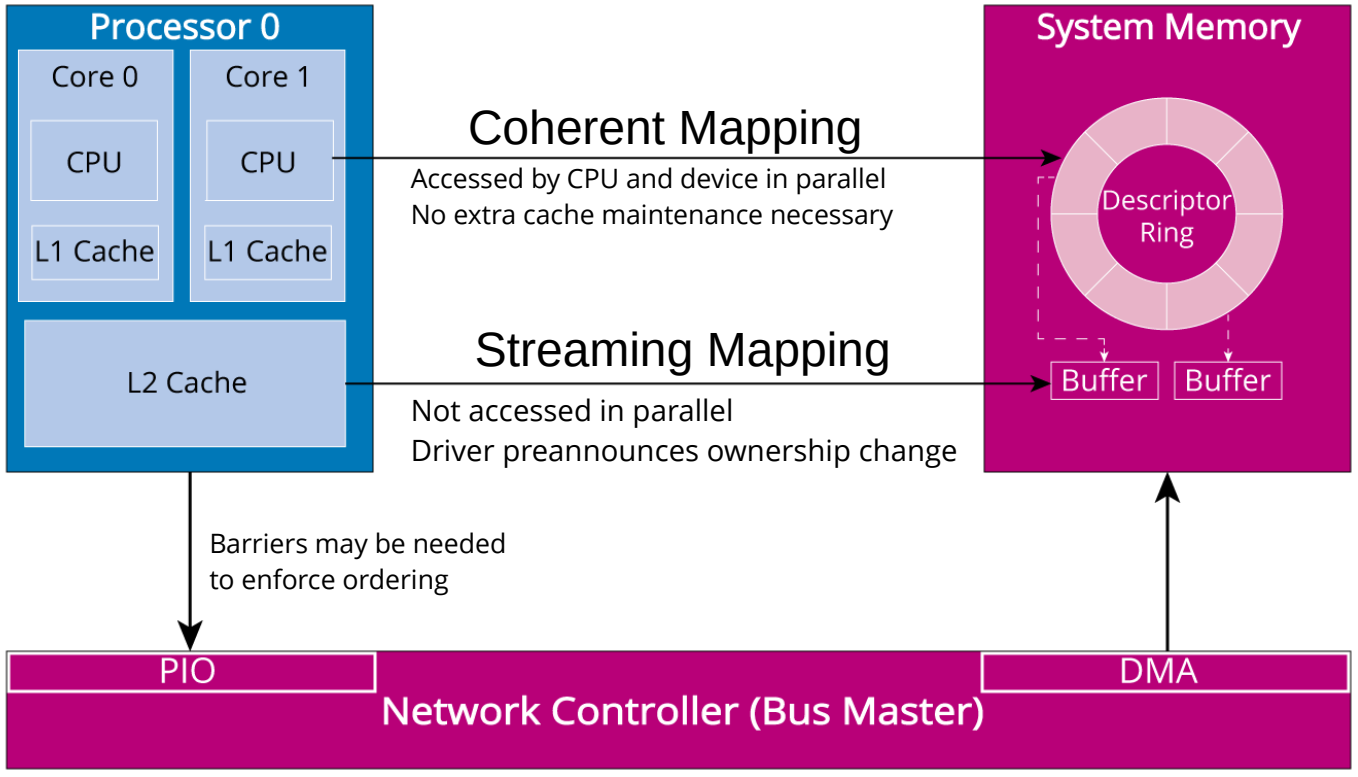
Cache-Coherent Interconnects

- Do devices mastering the bus snoop CPU local data caches?
  - Standard with x86
  - For ARM, depends on SoC interconnect:

    Some server grade uses it, most embedded SoCs don't
- For dma-noncoherent systems, Software must do cache maintenance before and after device DMA



DMA Mappings



DMA Mappings on dma-noncoherent ARMs

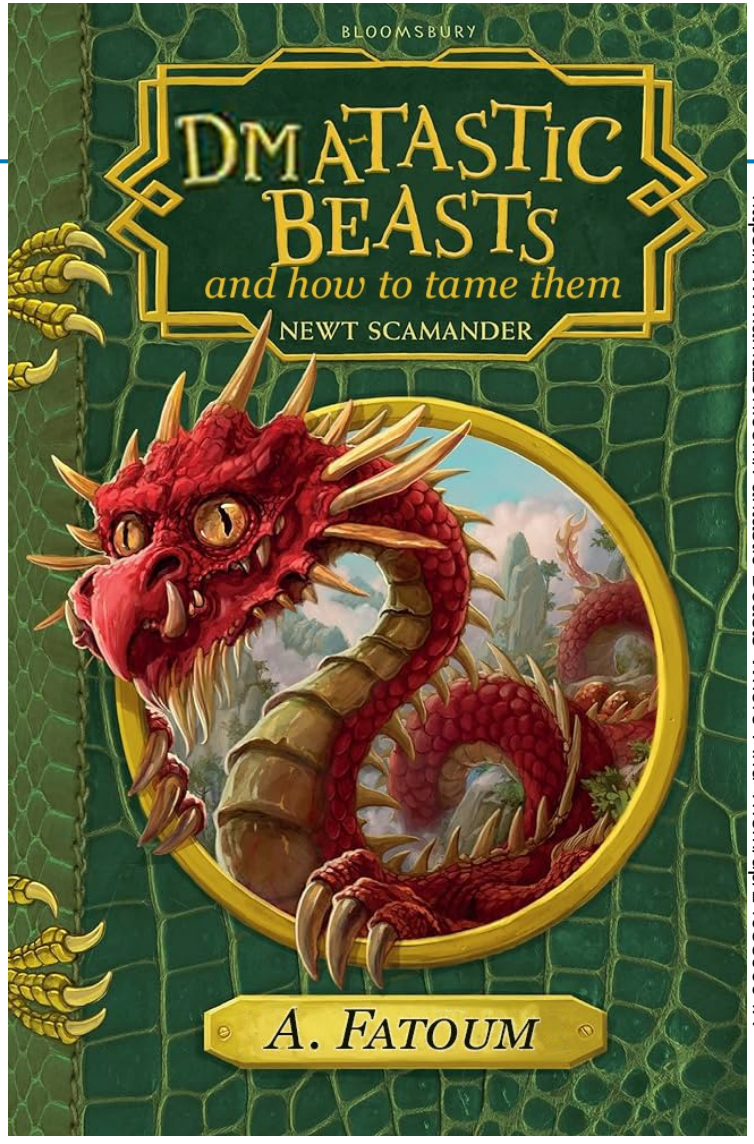
- Coherent DMA mappings
  - Page table has bits for caching attributes
  - `dma_alloc_coherent`: allocate and map uncached memory
- Streaming DMA mappings
  - `dma_map/dma_sync_for_device`
    - `DMA_FROM_DEVICE`: CPU invalidates (discards) caches
    - `DMA_TO_DEVICE/DMA_BIDIRECTIONAL`: CPU cleans (writes back) caches
  - `dma_unmap/dma_sync_for_cpu`
    - `DMA_FROM_DEVICE/DMA_BIDIRECTIONAL`: CPU invalidates caches
- Barriers to enforce ordering: more on that later

https://armstrongmanufacturing.com/product/clean-n-flush/

What we've talked about

- Why DMA?
- How DMA?
- What DMA?
  - We'll mostly talk about DMA API misuse in Linux and barebox.

Recommendation: "Mastering the dma and iommu apis" - Laurent Pinchart



https://www.amazon.com/Fantastic-Beasts-Where-Find-Them/dp/140889694X

📜 More RAM = More Crashes

- MMC host driver seemed to work fine
- System gets a RAM upgrade
- Spurious SMMU context faults, ADMA errors
- MMC controller itself apparently capable of full 64-bit addressing
  - Yet, issue disappears when limiting memory size to 3GiB

More RAM = More Crashes

- MMC host driver seemed to work fine
- System gets a RAM upgrade
- Spurious SMMU context faults, ADMA errors
- MMC controller itself apparently capable of full 64-bit addressing

  → But only 40 bits are wired to the bus resulting in address truncation:
  dma_set_mask_and_coherent(dev,
  DMA_BIT_MASK(40));

Two are better than one

- First TFTP transfer is corrupted

```
barebox@Linux Automation MC-1 SCMI board:/ md5sum /mnt/tftp/zImage
1e96da552d5c09f38d2c89c7cba765bd  /mnt/tftp/zImage
barebox@Linux Automation MC-1 SCMI board:/ md5sum /mnt/tftp/zImage
abcba100ed2b20bd70b4a1cfaacdad17  /mnt/tftp/zImage
barebox@Linux Automation MC-1 SCMI board:/ md5sum /mnt/tftp/zImage
abcba100ed2b20bd70b4a1cfaacdad17  /mnt/tftp/zImage
```

- But ownership seems correctly transferred to CPU

```
dma_sync_single_for_cpu(macb->dev, buffer, length, DMA_FROM_DEVICE);
net_receive(edev,
            macb->rx_buffer + macb->rx_buffer_size * macb->rx_tail,
            length);
dma_sync_single_for_device(macb->dev, buffer, length, DMA_FROM_DEVICE);
```

Two are better than one

- Ownership was indeed correctly transferred to CPU
- But before that, ownership was never transferred to device
  - A memory region is freed, but some of its dirty cache lines remain
  - The memory region is allocated again, this time as receive buffer
  - The dirty cache lines are still not invalidated
  - Packet written into memory by network controller via DMA
  - Stale dirty cache lines written back during eviction
  - Packet memory partially corrupted

Two are better than one

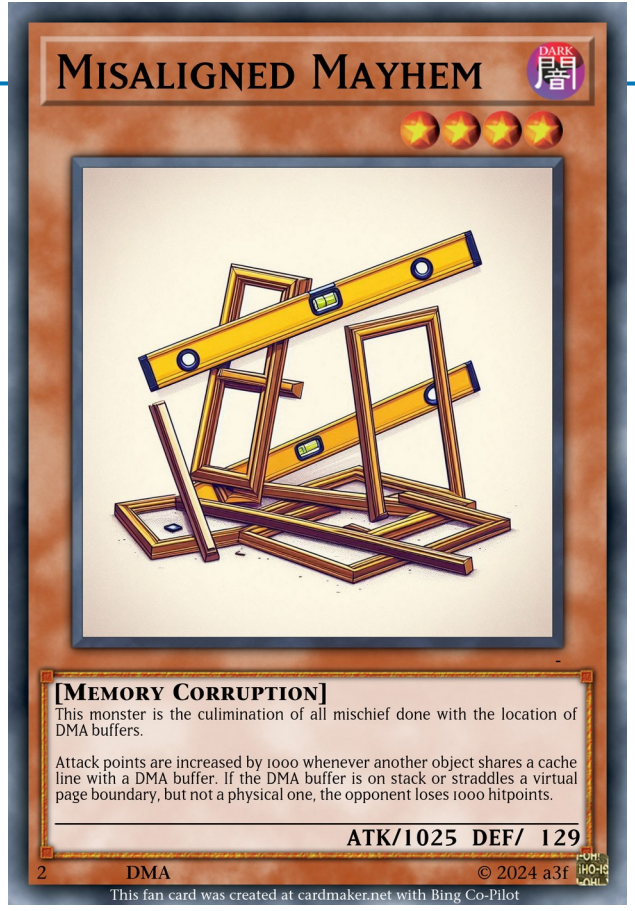
- Classic misuse of the API
  - Pointer casts & virt_to_phys are red flags
- Linux detects it when CONFIG_DMA_API_DEBUG=y
  - Now available in barebox too
- Straight forward fix:

```
@@ -388,0 +398,5 @@ static void macb_init(struct macb_device *macb)
-       macb->rx_buffer_phys = virt_to_phys(macb->rx_buffer);
+       macb->rx_buffer_phys = dma_map_single(macb->dev, macb->rx_buffer,
+                                             macb->rx_buffer_size * macb->rx_ring_size,
+                                             DMA_FROM_DEVICE);
+       if (dma_mapping_error(macb->dev, macb->rx_buffer_phys))
+               return -EFAULT;
```

- Added benefit: Support more than just 1:1-mapped devices

ⓘ Permissible memory for DMA

- Start is cache-line aligned: No sharing with previous buffer
- Size is cache-line aligned: No sharing with following buffer!
- Physically contiguous if it spans multiple pages (in absence of IOMMU)
- Addressable by the device (dma_mapping_error() != true):
  - Reduced physical address space
  - Not 1:1 view of the address space
  - Uncached aliases on the CPU side
- kmalloc + dma_map_single are the way to go
  - ARCH_DMA_MINALIGN / ARCH_WANT_KMALLOC_DMA_BOUNCE
  - GFP_DMA and GFP_DMA32 are likely not what you're after
- Take care when using same allocation for more than just one DMA buffer



📜 Flunking the Self-Test

- SoC Crypto Engine does self test on startup
  - Queue some descriptors
  - Wait for results in output rings
  - Match results with queued jobs
- Every now and then, self test fails and kernel panics
  - Address or other fields in output ring differ from submitted job

Flunking the Self-Test

- Turns out the device is cache coherent, but Linux didn't know

Flunking the Self-Test

- Turns out the device is cache coherent, but Linux didn't know
  - dma_map flushes cache lines
  - CPU prefetches buffer into cache
  - Device writes to buffer, write goes to CPU (write-back) cache, not external memory
  - dma_unmap invalidates cache line discarding received data in the process

Flunking the Self-Test

- Solution: Tell Linux with DT property that some devices are dma-coherent
- In the meantime, dma-coherent is now applied SoC-wide

Flunking the Self-Test

- Solution: Tell Linux with DT property that some devices are dma-coherent
- In the meantime, dma-coherent is now applied SoC-wide
  - Broke A/B boot
  - barebox now dynamically rewrites dma-coherent property

ⓘ DMA configuration inheritance

- DMA settings in device tree are inherited, but not in the driver model!
- Example:
  - DWC3 USB controller create virtual xHCI device to be bound by generic xHCI driver
  - Driver uses xHCI device for DMA API
  - dma-coherent property, in /soc or in the DWC3 USB controller has no effect!

📜 Reverse USB-Killer

- barebox newly ported to Rockchip RK356x
- DWC3 Gadget driver that worked fine on other platforms crashes in receive path

- Stack trace points out the offending instruction:
  - code calling memcpy looks identical to the kernel
  - memcpy in coherent memory with the address being 4-byte aligned

ⓘ barebox memory mapping

- Simpler than the kernel's at the cost of performance.
  In Linux if access order is important:
  - https://www.kernel.org/doc/Documentation/memory-barriers.txt
  - access to same MMIO device → volatile access in {read,write}X_relaxed
  - access to MMIO device then memory → barriers built into {read,write}X
  - access to coherent memory → dma_wmb/rmb/mb
  - access to coherent memory then MMIO → strict wmb/rmb/mb
- In barebox, MMIO and DMA coherent regions are strongly ordered
  - No need for memory barriers between accesses
    - writel_relaxed and writel are identical, just volatile accesses
  - Strongly ordered memory is not bufferable

Recommendation: " Uh-oh, It's I/O Ordering!" - Will Deacon

Reverse USB-Killer

- Optimized memcpy copies multiple bytes at once with unalignment handled in hardware
- Handling unaligned accesses to memory requires it to be mapped bufferable
- Solution: use memset_io/memcpy_(from/to)io

ⓘ Asking for a friend

- Not really a debugging story, but worth mentioning nonetheless
- Pretty popular way to circumvent CPU Address Space Isolation
- DMA Masters that are not themselves TrustZone aware need to
  - be restricted in what portions of the physical address space they may access
  - have accesses by non-secure world rejected if they have access to secure memory

📜 Corrupted SD-Cards

- Fancy NeoPixel LED string with stringent timing requirement
- Fiddle PWM and DMA peripherals from userspace
- File system on SD-Card is toast

Corrupted SD-Cards

- Fancy NeoPixel LED string with stringent timing requirement
- Fiddle PWM and DMA peripherals from userspace
- Picked wrong DMA channel
  - Turns out currently running kernel uses it
  - Level up from dynamic memory to persistent memory corruption

📜 Kernel hang on boot

- CONFIG_KASAN or memory watch points can't find the origin of the memory corruption
- Corrupted memory looks partially like network packet data

Kernel hang on boot

- CONFIG_KASAN or memory watch points can't find the origin of the memory corruption
- Corrupted memory looks partially like network packet data
- Bootloader (or kernel before kexec) didn't quiesce the device
  - All drivers configuring DMA need to disable it at remove/shutdown time

📜 DMA'ing a lot vs. lots of DMA

- Microcontroller as I/O expander
- Looooots of small packets (e.g. CAN)
- Throughput well beneath theoretical limit

DMA'ing a lot vs. lots of DMA

- Microcontroller as I/O expander
- Looooots of small packets (e.g. CAN)
- Throughput well beneath theoretical limit

  → DMA-Setup is not free
  - Synchronization overhead (cache maintenance, interrupt signaling)
  - Some DMA controllers are … peculiar
  - Some drivers dynamically choose whether to use PIO or DMA depending on transfer size

📜 *Not* this tall to ride

- barebox requests file over TFTP
- Ethernet frame sent is 129 bytes long
- tcpdump on the other end shows the last byte differs in 70% of cases

  → off-by-one in cache routine:

```
--- a/arch/arm/cpu/mmu_64.c
+++ b/arch/arm/cpu/mmu_64.c
@@ -285 +285 @@ void dma_inv_range(void *ptr, size_t size)
-       unsigned long end = start + size - 1;
+       unsigned long end = start + size;
```

Not this tall to ride

```
--- a/arch/arm/cpu/dma_64.c
+++ b/arch/arm/cpu/dma_64.c
@@ -9 +9 @@ void arch_sync_dma_for_device(void *vaddr, size_t size,
-       unsigned long end = start + size - 1;
+       unsigned long end = start + size;
```

- Patch is absolutely correct, but had to be reverted and postponed one monthly release
  - Drivers ported from Linux often use kmalloc, not barebox' dma_alloc(), but only the latter had cache line alignment in barebox
  - With fix applied: Memory allocator book keeping of unrelated buffers following misaligned buffer in same cache line is invalidated and thereby corrupted...

    → Make kmalloc a wrapper around memalign(*DMA_ALIGNMENT*
    - Wastes some memory, but there's plenty of DRAM anyway.
    - Linux had similar problems with devm_kmalloc.

📜 The Buffer Shambler

- SPI controller driver did DMA and diligently mapped buffers it's passed
- UBIFS mounting was broken

  → UBIFS allocates big buffers with vmalloc
  - Physical contiguity not guaranteed and thus not suitable for dma_map_single!
- Fixed since v5.4: MTD layer will generically use bounce buffers as appropriate

The Cache Clash

- Could we instead do DMA and transforms vmalloc memory into page-wise scatter gather?
  - Not without the caller doing cache maintenance if needed
    - flush_kernel_vmap_range
    - invalidate_kernel_vmap_range
  - This is what XFS is doing.

Summary

- The Linux DMA API serves its purpose well: Portable handling of DMA in drivers across many different CPUs and architectures
- API has to work with what it's given: unannotated addresses without lifetime and ownership metadata
- CONFIG_DMA_API_DEBUG is great, but:
  - Debugging option not usually enabled in production kernels
  - Can't protect against everything

Recommendation: "DMA Safety in Buffers for Linux Kernel Device Drivers" - Wolfram Sang

Future Outlook

- Rust's more expressive type system could eliminate classes of misuse around DMA in drivers at compile-time
- CONFIG_DMA_API_DEBUG could be improved for C by poisoning DMA buffers with KASAN while owned by device
- → barebox PoC: dma: debug: track DMA buffer ownership with KASAN
  - CONFIG_KCSAN_REPORT_RACE_UNKNOWN_ORIGIN=y can detect DMA races
- I/O MMUs should make life easier; Hoping they become ubiquitous in future embedded SoCs
- For barebox:
  - Fix remaining CONFIG_DMA_API_DEBUG issues
  - remove pointer casts and use DMA mapping API everywhere

Questions?